Orthogonal Persistence: The Future for Storing Objects?

Tony Printezis
Department of Computing Science
University of Glasgow
17 Lilybank Gardens
Glasgow G12 8RZ
Scotland
tel. +44 141 330 6044 / fax +44 141 330 4913
tony@dcs.gla.ac.uk

Abstract

One of the most tedious tasks for programmers is to write and maintain code for flattening and unflattening data structures in order to move them to and from long-term storage. This flattening and unflattening is necessary since the data model of the storage system (e.g. flat file, relational database) is traditionally very different to the one of the programming language (e.g. record, object). For example, flat files are structured as streams of bytes and program data structures have to be manipulated in order to be stored as such.

Recently, the popularity of the Java™ language has stirred an interest in exploring more intuitive and automated ways to store and retrieve data, which are better suited to the object-oriented programming model. The resulting solutions, e.g. object serialisation, mappings to object-oriented and relational databases, achieve this to a certain extent. However, it can be argued that any solution which involves two separate systems cannot be as flexible and efficient as a fully integrated one.

Orthogonally persistent systems have existed since the early 80s. They provide an intuitive and transparent model of making data persistent by completely integrating the persistence mechanism inside the language runtime system.

This paper presents the advantages of such systems over the traditional approaches and how they can improve programmers' efficiency and productivity. The Persistent Java (PJama) system, the result of a collaboration between Glasgow University's Computing Science Department and Sun Microsystems' Laboratories West, will be used as a concrete example.

1 INTRODUCTION

In most programming languages, the data structures created by a program are transient, i.e. they disappear at the end of the program execution and have to be re-created every time the program is re-run. The ability to retain some data across program executions is called *Persistence*.

Persistence is typically achieved with the use of a database. However, the traditional model of creating persistent programs, illustrated in Figure 1, is awkward for the programmers since they have to deal with three different entities and the interactions between them. The PJama project, a collaboration between Glasgow University's Computing Science Department and Sun Microsystems' Laboratories West, adopts the concept of orthogonal persistence for the popular Java™ language.

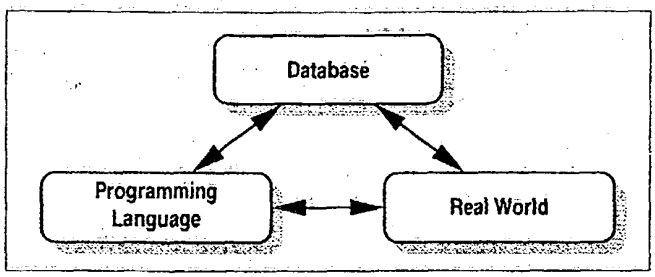

Figure 1: Traditional Programming Model.

It attempts to decrease the number of entities, which the programmer has to deal with, from three to two, by including the persistence facilities inside the language runtime system, while leaving the language itself unchanged.

Section 2 gives an introduction to the concepts of orthogonal persistence, Section 3 presents the goals of the PJama project, and finally Section 4 includes the conclusions and future work.

2 ORTHOGONAL PERSISTENCE

Orthogonal Persistent is a language-independent model of persistence, defined by the following three principles [8, 18].

1. Type Orthogonality
2. Persistence by Reachability
3. Persistence Independence

These are explained in more detail below.

2.1 Type Orthogonality

> *"Persistence is available for* all *data, irrespective of type."*

Many systems claim that they provide orthogonal (sometimes they call it *transparent*) persistence. However, deeper investigation usually reveals that rather than allowing "any data type" to persist, they allow "any data type, provided *X*", where *X* is: the data type has mapping/unmapping facilities defined on it, it is a subclass of a persistent-object class, it implements a certain persistent-related interface, etc. We strongly believe that this is not orthogonal persistence, and definetely not transparent, since the application programmer has to decide for *each* data type whether it is allowed to persist or not and, in some cases, has to write the mapping code explicitely.

2.2 Persistence by Reachability

> *"The lifetime of all objects is determined by reachability from a designated set of root objects, the* Persistent Roots."

The concept of reachability is well understood by anybody who uses a programming language which relies on a garbage collector for its memory management[1]. It is only natural to extend this concept to also apply to persistence and provide a uniform model for short and long lived data. There are persistent systems which, even though they target a garbage-collected language, require explicit deletes in order to reclaim space in the storage system. This forces the programmer to use two different models of storage management and we strongly believe that it severely minimises the benefit of introducing a garbage collector.

Persistence by reachability is also referred to as *Transitive Persistence*.

2.3 Persistence Independence

> *"It is indistinguishable whether code is operating on short-lived or long-lived data."*

It is very often the case that, when programmers use similar data structures in memory and on disk, they have to replicate their code and keep one version tied to the required storage system. This has the disadvantages of the programmer having to maintain two versions of the code, the API of the storage system "getting in the way" of the algorithm which is being implemented, and, if a different storage system needs to be targetted, yet another version of the code needs to be created. Additionally, any third-party libraries have to be modified in order to operate over persistent data. In fact, this situtation gets worse since the transient and persistent versions of the code might not be possible to co-exist in the same system (e.g. due to naming problems). The persistence independence principle allows for exactly the same code to operate over transient and persistent data. This deals with the problems mentioned above and can minimise development time, while maximising code re-use.

2.4 Benefits of Orthogonal Persistence

Given the above three principles, the interaction between the programmer and the orthogonally persistent system is minimal. In fact, apart from the usual facilities that a programming language provides, the only extra persistence-specific calls needed are the following.

- *Register and retrieve the persistent roots* (described in Section 2.2).

  We should emphasise that a persistent root should *not* be associated with small object graphs, but rather with entire applications. Therefore, the calls which deal with persistent roots are invoked very rarely (typically, only inside the application-startup code).

[1] For more information on garbage collection and its benefits, the book by Jones, the only one specialised on the subject, is a very good resource [16].

- Perform a checkpoint.

  This means that any changes performed on the data are atomically propagated to the disk and, in the event of a failure, they will be retrieved upon startup in a consistent state.

Another important benefit, that the architecture of orthogonally persistent systems usually allows, is the incremental on-demand object-fetching. This allows them to be much more responsive, when they are initialised, and eliminates the "Big-Inhale" problem, i.e. the initialisation of a system being really slow since it has to first fetch all the data it needs.

It is interesting to note here that, from past experience, it is usually fairly difficult to explain to experienced programmers the concept of orthogonal persistence. This is because a traditional database involves a mapping between programming language data structures and the database model and complex APIs through which the data is transferred. This is the model that seasoned programmers are used to and they get disappointed when they ask "What the API does look like?" or "How do I run queries?", since the API is minimal and queries are not directly supported, as query engines should be built at the application-level, possibly using standard bulk-type libraries. In fact, it is reported by Liedtke that novice programmers absorb much more easily the concepts of orthogonal persistence [21].

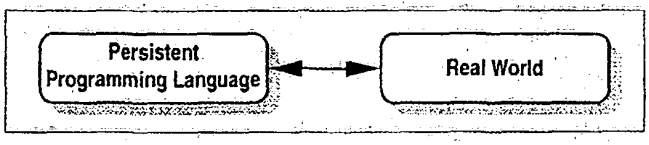

Figure 2: Orthogonally Persistent Programming Model.

The programming model of orthogonal persistent systems is illustrated in Figure 2. It only requires one mapping (from the "real world" to the persistent programming language), which is a big improvement, compared to the three needed in traditional systems (as illustrated in Figure 1).

2.5 A Historical Note

Orthogonal persistence was first proposed by Atkinson in 1978 [3] and its benefits are described in detail by Atkinson and Morrison [8].

The first persistent programming language is considered to be Pascal/R [28], an extension of Pascal which allowed relational queries inside the language. However, the first orthogonally persistent language was PS-Algol [4], an extension of S-algol [23], developed by the Universities of Edinburgh and St Andrews, Scotland, around 1980.

PS-Algol was succeeded by Napier88 [24], developed at the University of St Andrews, Scotland, which had a much richer type system and introduced parametric polymorphism. By this point, increased interest in orthogonally persistent systems yielded the languages Fibonacci [1], developed at the University of Pisa, Italy and Tycoon [22], at the University of Hamburg, Germany. In fact, the above three systems were developed as part of the FIDE (Fully Integrated Data Environments)

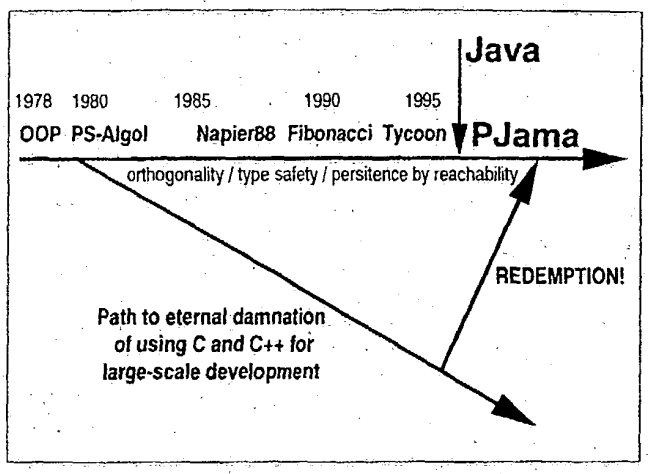

Figure 3: The Java event and orthogonally persistent systems.

and FIDE$_2$ ESPRIT projects. A book is being written on the contribution of these two projects and it is about to be published [9].

It was only natural for the popularity of Java [15, 2] and the benefits of orthogonal persistence to be combined in the form of the PJama project, which is described in Section 3. Figure 3 shows a timeline which follows the development of the systems mentioned above and current trends in softward development and how the "Java event" affected it.

3 PJAMA: AN ORTHOGONALLY PERSISTENT JAVA

PJama [7, 5] is a system which provides orthogonal persistence for the Java programming language [15, 2]. It was developed collaboratively between the Department of Computing Science of the University of Glasgow and the Research Laboratories of Sun Microsystems. It conforms to the three principles of orthogonal persistence, described in Section 2, since

- instances of *any* class can persist (this includes the classes themselves, as they are instances of the class `Class`, and their static fields, GUI components, etc.),
- all objects reachable from a set of declared roots become persistent, and
- code which operates over transient data can also operate over persistent data, with no changes to the original source or post-processing of the bytecodes being necessary.

Unfortunately, our claim of complete type orthogonality is not yet entirely true. Even though the majority of classes can persist unchanged, there is a small number of them which either require some changes in order to persist or cannot persist at all. Examples of classes which require changes are ones which use the `transient` keyword in a manner incompatible with orthogonal persistence

(this is explained in more detail by Printezis, Jordan, and Atkinson [25, 18]) or depend on static initialisers to load dymanic libraries (as explained below, in Section 3.1). Examples of classes which cannot persist at all are ones tied closely to the implementation of the VM (`Thread`, `Exception`, etc.).

The issues of type orthogonality in Java are discussed in more detail by Jordan and Atkinson [17, 18, 6]. Currently, most of the obstacles which are in our way to complete orthogonality are technical issues and we remain committed to overcome them in the near future.

3.1 Making Classes Persistent

We believe that it is imperative to keep the classes in the store along with the data. This way, the application programmer does not have to manually keep track which version of a class matches the persistent data in the store, which can be complex and tedious for large projects. Instead, we prefer to rely on our own class-evolution tools to evolve the classes in the store, safely and consistently. Even though, this is a hard and complex problem, we are making steady progress on it, as reported by Dmitriev [12].

A class `C` becomes persistent if one of its instances becomes persistent or if it is used by another class `D` which has become persistent (e.g. if `D` accesses one of `C`'s static methods or fields). When a class becomes persistent, its static fields also become persistent. If this was not the case, then any state in the static fields, which is shared by all instances, would be lost; this would force the class implementation to change, breaking the concept of persistence independence.

The only time that the static initialiser of a class is called is when the class is loaded from the file system. However, if it becomes persistent, the static initialiser is *not* called every time the class is fetched from the store, otherwise its static fields will be re-initialised and their values might be inconsistent with the state of the persistent instances of that class. If some initialisation needs to be performed every time a class is fetched (e.g. dynamic library loading), this can be done by the *Action Handlers* which PJama defines [31, 18].

3.2 The PJama Architecture

PJama achieves the three principles of orthogonal persistence, mentioned in Section 2, by requiring changes to the Java Virtual Machine (JVM). It can be argued that this is the only way to make some classes persistent (e.g. `Class`, `Thread`, etc.), since their state cannot be accessed from Java and therefore a Java-only solution would be inappropriate. In fact, GemStone have taken the same approach for their GemStone/J product.

The current PJama implementation is based on Sun's JDK. A very high-level illustration of its architecture is given in Figure 4. The original JDK comprises the JVM and the *Transient Heap*[2] (TH), where objects are allocated and manipulated. PJama extends this by adding the *Object Cache* (OC), where persistent objects are cached and manipulated. Objects in the TH and the OC appear

[2]This is also known as the *Garbage Collected Heap*.

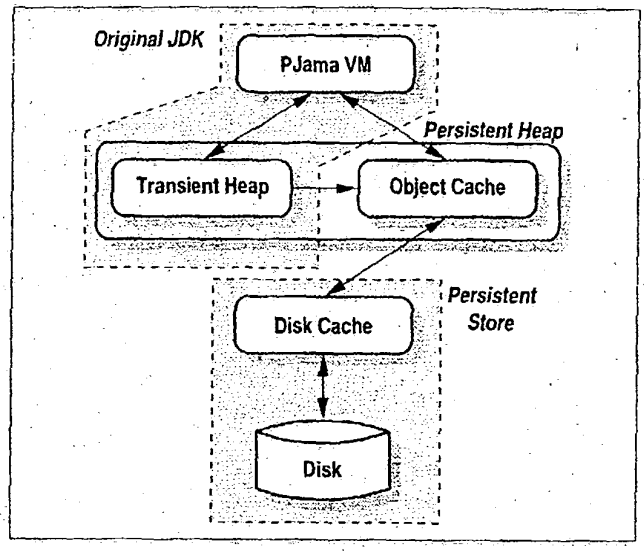

Figure 4: The PJama Architecture.

the same to the JVM, therefore the combination of the TH and the OC can be viewed as a single *Persistent Heap*. In fact, these two components might be unified in future high-performance implementations of PJama.

When an object in the TH becomes persistent (i.e. when there is a reference to it from a persistent object in the OC), it is moved to the OC and an image of it is written to disk, via the *Disk Cache* (DC) of the persistent store. When an object needs to be fetched from disk, the buffer where it resides is copied to the DC and then the object is copied to the OC. When the OC fills up, objects are automatically evicted in order to make space for newly-fetched ones. More information on the memory management of PJama is given by Daynès and Atkinson [11].

The above architecture allows the performance of the PJama system to be very good, since all the object-fetching and dirty object-tracking is done entirely inside the runtime system and not in Java. In fact, some performance evaluation experiments by Ridgway *et al.* show PJama faster than all the Java-only approaches which map objects to relational or object-oriented databases [27].

3.3 The PJama API

The PJama API is presented in Figure 5. For simplicity and clarity, the exception-related lines have been omitted. All the necessary calls are declared in an interface called `PJStore`. The motivation behind this is to allow any third parties, who work on orthogonally persistent systems for Java, to use the same API and allow code to be portable between different systems. As explained in Section 2.4, the API is very minimal and comprises of a few calls to manage persistent roots, the `stabilizeAll` call, which performs the checkpoint operation mentioned in Section 2.4, and a few

```
package org.opj.store;
import java.util.Enumeration;

interface PJStore {
  public void newPRoot(String, Object);
  public Object getPRoot(String);
  public void discardPRoot(String);
  public boolean existsPRoot(String);
  public Enumeration getAllPRootNames();

  public void stabilizeAll();
  ...
}
```

```
package org.opj.store;

class PJStoreImpl implements PJStore {
  static private PJStore store;
  static public PJStore getStore() {
    return store;
  }
  ...
}
```

Figure 5: The PJama API.

other utilities, which are omitted from the figure.

Our implementation class is called `PJStoreImpl` and is also included in Figure 5. It implements the `PJStore` interface and it has a static field called `store`, which represents the persistent store. This is initialised when the JVM is initialised and can be accessed by the programmer with the `getStore` static method. A concrete example of how the API is used is given in Section 3.4.

Currently, all of the PJama API is included in a package called `org.opj`. However, our ambition is to include these facilities in the standard Java libraries. This is specified in the Orthogonal Persistence for Java (OPJ) Specification document [19].

3.4 A Concrete Example

Figure 6 illustrates a concrete example of using PJama. Again, for simplicity and clarity, any exception-related lines have been omitted. Notice that all the PJama-specific code is included in grey boxes.

The class `ExtHashtable` is a subclass of the standard class `Hashtable` and introduces two new methods: `populate`, which populates the hash table in some manner (e.g. by reading a file or relying on user-input), and `display`, which simply displays the contents of the hash table.

The class `TransientCreateRead` illustrates how `ExtHashtable` is used. It simply creates an instance of `ExtHashtable`, populates it, and displays its contents. Of course, as in any transient program, the contents of the hash table cease to exist once the program finishes execution.

The class `PersistentCreate` illustrates how we can make the contents of the hash table persistent, using PJama. After the hash table has been instantiated, it is registered as a persistent root, asso-

```
import java.util.Hashtable;

class ExtHashtable extends Hashtable {
  void populate() {
    /* populates the hash table */
  }

  void display() {
    /* displays the contents of hash table */
  }
}
```

```
import org.opj.store.*;

class PersistentCreate {
  static public void main (String args[]) {
    PJStore PS = PJStoreImpl.getStore();
    ExtHashtable ht;

    ht = new ExtHashtable();
    PS.newPRoot("Hash Table", ht);
    ht.populate();

    /* implicit stabilisation */
  }
}
```

```
class TransientCreateRead {
  static public void main (String args[]) {
    ExtHashtable ht;

    ht = new ExtHashtable();
    ht.populate();
    ht.display();
  }
}
```

```
import org.opj.store.*;

class PersistentRead {
  static public void main (String args[]) {
    PJStore PS = PJStoreImpl.getStore();
    ExtHashtable ht;

    ht = (ExtHashtable)
      PS.getPRoot("Hash Table");
    ht.display();
  }
}
```

Figure 6: A concrete example of using PJama.

ciated with the name "Hash Table", and then populated. Since PJama has an implicit checkpoint operation at the end of program execution, it will automatically propagate all persistent objects, in this case, the hash table and its contents, to the persistent store before exiting.

The class `PersistentRead` illustrates how persistent data can be retrieved, using PJama. The persistent root called "Hash Table" is looked-up and this yields a reference to the `ExtHashtable` object. This can then be manipulated in the normal manner. It is worth pointing out here that, at the point when the persistent root is retrieved, the entire hash table is *not* fetched into memory; only the parts of it which will be accessed are fetched, as and when are needed.

In each example above notice that only three lines are PJama-specific (plus some exception handling, which is not illustrated) and, in both cases, they are incorporated in a single class, leaving the original `ExtHashtable` class, which could have been obtained from a third-party, unchanged.

3.5 Comparisons with Other Systems

Currently, there is a large number of systems which provide persistence for Java. A lot of them are provided in the form of extended APIs and their implementation tends to be written either purely in Java or mostly in Java, with a small set of native methods through which the communication with the storage mechanism takes place. We believe that inherently such systems can neither achieve full orthogonality nor be as flexible as PJama.

The first persistence-related API, developed by Sun Microsystems, was JDBC™, an API which allows access to relational databases via SQL queries [30]. Obviously, this approach, even though convenient when accessing legacy data, is far from transparent or even portable, since it is well-known that different relational database vendors implement their own version of SQL.

Since the 1.1 release of Sun's JDK, *Java Object Serialisation* (JOS) [29] has been considered to be the default persistence mechanism for the Java language. JOS is now part of the standard `java.io` package of Java and provides facilities to serialise/de-serialise an object graph to and from a byte-stream. The generated byte-stream can either be written to disk or sent to another machine over the network (the latter being the original use of JOS). We believe that there are numerous problems with JOS being used to provide production-quality persistence for Java (individual objects on the byte stream cannot be updated but the byte-stream has to be re-created instead, it suffers from the big-inhale problem, described above, as the byte-stream must be entirely de-serialised before any objects can be used, etc.); these are summarised by Evans [13]. However, our biggest criticism is that JOS is not orthogonal, since for an instance to be allowed to be serialised, its class must implement the `Serializable` interface. Many of the core classes do not do this. This destroys both the type orthogonality and the persistence by reachability principles.

Another category of systems which provide persistence for Java includes ObjectStore PSE [20] (written entirely in Java) and the systems which conform to the ODMG standard [10] for object-oriented databases (such as systems from Versant, Ardent, Poet, etc.). These also claim to implement orthogonal persistence by reachability. However, the way this is achieved is by post-processing the Java class files to include "transparent" extra calls to the persistence-related classes. This results in requiring two versions of the classes (one for transient and one for persistent use), which complicates their management and introduces extra complexity to the programmer. For example, the popular JGL collection classes are available in two versions, standard and "annotated" for use with ObjectStore PSE. We strongly believe that, if this trend spreads further, it will become intolerable. Furthermore, these systems actually do have limitations on which classes they will allow to persist. An example of this is ObjectStore PSE, which provides its own hash table implementation, since it has problems dealing with the standard `java.util.Hashtable` class.

Finally, the system which is probably closest to PJama, according to persistence model, flexibility, and performance, is the GemStone/J product from GemStone [14]. This is the case, since GemStone have taken a similar approach to the PJama project and require a special persistence-aware JVM, rather than attempting a Java-only solution. It is based on their previous technology for SmallTalk and supports both class evolution and efficient disk garbage collection.

4 CONCLUSIONS AND FUTURE WORK

This paper has presented the benefits of orthogonal persistence in the context of the Java language and the current implementation of the PJama system has proved its feasability. The current stable release of the system, available for SPARC and x86 Solaris, is based on JDK1.1.7, but we have made a pre-release based on the latest JDK1.2, which we are still improving. It can be downloaded, for evaluation purposes, from the following Web site

```
http://www.sunlabs.com/research/forest/opj.main.html
```

where additional information on the project can also be obtained.

PJama can currently handle up to 2GB stores, it delivers its promise of type orthogonality, since it can handle most third-party classes (see Section 3 for the exceptions), it can provide persistence for the Java SwingSet GUI components, and it ships with simple but functional tools for persistent store garbage collection and class evolution.

Our future plans include a PJama port to a high-performance Java virtual machine, integration with Sphere [26], the new persistent store developed at the University of Glasgow, which provides support for incremental disk garbage collection and class evolution, and progress towards complete type orthogonality. Finally, we will carry on campaining for orthogonal persistence to be included in the core of the Java language.

ACKNOWLEDGEMENTS

The author would like to thank Prof. Malcolm Atkinson and Robert Greig for their useful and constructive comments on this paper. These days, large software projects can only be built by teams of programmers and PJama is no exception. The author is grateful to all the members of the PJama team, past and present. In particular, he would like to thank Prof. Malcolm Atkinson and Dr Mick Jordan, who head the teams at Glasgow and SunLabs West respectively, and Susan Spence, Craig Hamilton, Laurent Daynès, Brian Lewis, Michael Van De Vanter, Bernd Mathiske, Neal Gafter, and Misha Dmitriev for their support and input to this research.

REFERENCES

[1] A. Albano, G. Ghelli, and R. Orsini. An Introduction to Fibonacci: a Programming Language for Object Databses. In Atkinson and Welland [9], chapter 1.1.2. To be published.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] M. P. Atkinson. Programming Languages and Databases. *VLDB Journal*, pages 408–419, 1978.

[4] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[5] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record*, December 1996.

[6] M. P. Atkinson and M. J. Jordan. Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java[tm]. In *Proceedings of ICDT'99*, Haifa, Israel, January 1999.

[7] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: a Type-Safe Object-Oriented Orthogonally Persistent System. In *Proceedings of POS'7*, Cape May, New Jersey, USA, May 1996.

[8] M. P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[9] M. P. Atkinson and R. Welland, editors. *Fully Integrated Data Environments*. Springer-Verlag, 1999. To be published.

[10] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[11] L. Daynès and M. P. Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[12] M. Dmitriev. The First Experience of Class Evolution Support in PJama. In *Proceedings of the Third International Workshop on Persistence and Java (PJW3)*, Tiburon, California, September 1998.

[13] H. Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, Department of Computing Science, University of Glasgow, Scotland, 1999. *In Preparation.*

[14] GemStone Systems Inc. *GemStone/J2.0: The First EJB Server Based on CORBA and an OTM*, 1998.

[15] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[16] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[17] M. J. Jordan. Early Experiences with Persistent Java. In *Proceedings of the First International Workshop on Persistence and Java (PJW1)*, Drymen, Scotland, September 1996.

[18] M. J. Jordan and M. P. Atkinson. Orthogonal Persistence for Java — A Mid-term Report. In *Proceedings of the Third International Workshop on Persistence and Java (PJW3)*, Tiburon, California, September 1998.

[19] M. J. Jordan and M. P. Atkinson, editors. Orthogonal Persistent for the Java[tm] Platform — Draft Specification. Technical report, Sun Microsystems Inc, 1999. *In Preparation.*

[20] G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. ObjectStore PSE: a Persistent Storage Engine for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[21] J. Liedtke. A Persistent System in Real Use — Experiences of the First 13 Years. In *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems*, 1993.

[22] F. Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In Atkinson and Welland [9], chapter 1.1.1. To be published.

[23] R. Morrison. S-algol Language Reference Manual. Technical Report CS/79/1, University of St Andrews, 1979.

[24] R. Morrison, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, D. S. Munro, and M. P. Atkinson. The Napier88 Persistent Programming Language and Environment. In Atkinson and Welland [9], chapter 1.1.3. To be published.

[25] T. Printezis and M. P. Atkinson. Defining and Handling Transient Data in Persistent Systems, 1999. *In Preparation.* To Be Submitted for publication at *DBPL'99*, Kinloch Rannoch, Scotland.

[26] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of a new Persistent Object Store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[27] J. V. E. Ridgway, C. Thrall, and J. C. Wileden. Towards Assessing Approaches to Persistence for Java™. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[28] J. W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of SIGMOD'77*, August 1977.

[29] Sun Microsystems Inc. *Java™ Object Serialization Specification*, November 1998. Revision 1.43.

[30] Sun Microsystems Inc. *JDBC™ 2.0 API*, May 1998. Version 1.0.

[31] Sun Microsystems Inc and The University of Glasgow. *PJama API*, 1998. Release 0.5.7.13.